Computer-implemented method and system for automatically invoking a predetermined debugger command at a desired location of a single thread of a program

FIELD OF THE INVENTION

This invention relates to computer program debuggers.

BACKGROUND OF THE INVENTION

Program debugging is done in one of the following two ways. According to a first approach, a debugger is used, a debugger being a tool that enables a partial execution of the program, stopping at predefined points such as lines of code or variable values. A second way is log-based debugging, wherein print statements are added to the program. When a test is executed a log is created. This log is examined off-line by the programmer to look for places where the behavior of the program deviates from the expected behavior. There also exist debugging tools that display traces of executions and show what happens during the execution. Once the location of a problem is found using these tools, other tools such as a debugger are used to debug the program. It is not possible, after identifying the location of the problem, to return to the actual program to investigate the state of the program. Getting to the correct location using a debugger can be a difficult problem, because algorithmic debugging, i.e. locating a bug, is difficult once a fault occurs.

Sub A1

Algorithmic debugging is used to support locating a defect once a fault is found, as described in web site http://www.cs.nmsu.edu/~mikau/aadebug.html and "*Generalized algorithmic debugging and testing*" by Peter Fritzson *et al.* appearing in "*ACM Letters on programming languages and testing*", 1:303-322, 1992.

There are many existing debugging tools and there are several trace visualization tools available, but there are no tools that combine the use of debugging commands and trace control together.

SUMMARY OF THE INVENTION

It is therefore an object of the invention to provide a method and system for automatically invoking a predetermined debugger command at a desired location of a single thread of a program.

Sub A2

To this end, there is provided in accordance with a first aspect of the invention a computer-implemented method for automatically invoking a predetermined debugger command at a desired location of a single thread of a program containing at least one thread, said method comprising:

(i) embedding within said single thread at said desired location thereof a utility which reads a trace file in which said predetermined debugger command has been previously embedded; and

(ii) running the program for reading said trace file and invoking said predetermined debugger command.

The method according to the invention thus combines log-based debugging and use of a debugger. By such means, it is possible to examine the log and then to start the program, using the debugger at a point selected by the user. Such an approach may be implemented on large, multi-threaded programs containing hundreds of threads.

Implementation is especially difficult in the context of multi-threaded programs, owing to the fact that execution may result in different interleaving each time the program is run. That is to say that the mutual sequence in which more than one thread perform their instructions may not be constant, since from each thread's point of view, all that is important is that it performs its own instructions in the correct sequence providing that the integrity of the values of variables as read by the thread is not compromised. Such integrity is assured by the replay mechanism as explained below. It does not matter if that sequence is interrupted to allow another thread to perform instructions. However, this may have an effect on correct generation of the trace file.

The solution according to the invention requires several components:

(א) an instrumentation scheme, which allows the user to put specialized print statements in the program, both manually and automatically, to create trace files.

(ב) a modified debugger that can be executed "against" the trace files. This debugger, when encountering an instrumentation statement, will check the trace file. If the trace file contains a debugger command (such as show current program status) it will execute it, else it will continue if appropriate. This is the appearance to the user. Implementation may be done without modifying the debugger.

(ג) a number of supporting algorithms used to re-execute the program with the same multi-threading interleaving, and, if needed, to create a naming scheme for threads or to match between threads and traces.

BRIEF DESCRIPTION OF THE DRAWINGS

In order to understand the invention and to see how it may be carried out in practice, a preferred embodiment will now be described, by way of non-limiting example only, with reference to the accompanying drawings, in which:

Fig. 1 is a flow diagram showing the principal operating steps carried out by a method according to a first embodiment of the invention for invoking a debugger from a trace file;

Fig. 2 is a flow diagram showing the principal operating steps carried by a **"Trace-Print"** function inserted into program code in the method shown in Fig. 1;

Fig. 3 is a flow diagram showing the principal operating steps carried by a scheme for automatic naming of trace files as a function of a creating thread's name;

Fig. 4 is a pictorial representation of a tree structure depicting a scheme for automatic naming of trace files as a function of a creating thread's name; and

Fig. 5 is a block diagram showing functionally a system for carrying out the invention.

DETAILED DESCRIPTION OF THE INVENTION

Fig. 1 is a flow diagram showing the main operating steps carried out by a method according to a first embodiment of the invention for invoking a debugger from a trace file. In common with known tracing techniques, print statements are inserted into the program source code so as to output the value of a specified variable at each location in the program where the value of the variable is significant and where it may be important subsequently to invoke a debugger command. However, unlike known tracing techniques, the print statements that are inserted into the program source code are a modified print function,

"Trace-Print", that operates differently according to whether or not a trace file already exists. Specifically, the modified print function creates a trace file if the file does not exist, and writes the value of the variables thereto.

This having been done, a programmer takes the trace file and reviews it line by line. When he or she comes across a value which appears suspect, a debugger command is either inserted into the trace file before the line in the trace file corresponding to the suspect line or, alternatively substitutes the line containing suspect variable with a debugger command.

This having been done, the program is re-run. It is assumed that the program retraces its previous steps as recorded in the trace file until it reaches the first modified print function.

Fig. 2 shows the principal operating steps carried by the modified print function, **"Trace-Print"**. Thus, as noted above, on determining that the file does not exist, the **Trace-Print** function creates a trace file and writes the value of the variables thereto. However, if the trace file does already exist, then the current value of the variable output by the program at the current location of the **Trace-Print** function is compared with a respective line in the trace file. If they are different, the **Trace-Print** function construes the respective line in the trace file as a debugger command and invokes a debugger so as to execute the debugger command.

A particular example follows wherein different components of the invention are described in greater detail.

Instrumentation:

Functions of the form **Trace-Print** (Trace-Name, Content) are added to the program. Preferably, this is done automatically, using a menu, in all locations of a

specific type defined by the programmer (e.g. beginning of routines, places where a particular variable "X" is used, etc.). In the case of multi-threaded programs, automatic choice of the trace granularity ensures replay ability. Alternatively, the **Trace-Print** functions may be added to the program manually. Replay ability is described in "*Deterministic Replay of Java Multithreaded Applications*", by Jong-Deok Choi and Harini Srinivasan, in the Proceedings of the ACM SIGMETRICS symposium on parallel and distributed tools, 1998.

When the program is executed on a test, a flag called "**compare**" is added. If **compare**=*false*, then the **Trace-Print** function prints Content into file **Trace-Name** in the form "**Trace: Content**".

If **compare**=*true*, then the program is executed under a modified debugger.

Modified Debugger:

Whenever a **Trace-Print** function is reached, if the content of Trace-Name is of the form "**Debug: Command**", then this debugger command is executed. As shown in Fig. 2, such execution can be performed either iteratively or in batch mode.

IF the content is of the form "**Trace: Content**", **THEN**

IF the content in the trace file equals **Content** of **Trace-Print**, then continue;

ELSE raise the debugger at this point with the appropriate error message.

Alternatively, the function **Trace-Print** is modified such that if the file **Trace-Name** is one of the debug-trace file names it either writes to the file as above or compares, much like the previous implementation but implemented within the program, possibly as a library. When a point is reached where the debugger is needed, it is invoked from within the program. This implementation

will work only for debuggers that can be invoked while the program is running, as is the case with most debuggers. The advantage of this approach is that the debugger is not modified, and it is easy to port between debuggers.

Alternatively, a new application may be created that does not run under the debugger. Such an application accepts as input a trace file and a stream of outputs from the executing program and compares the stream to the trace file. If the stream is different, it halts the program and raises a debugger. Such an approach has the advantage that the user program is not modified and the application is not language specific. In either case, application of the debugger gives rise to delays and it is therefore hard to break the program exactly at the desired location.

One possible solution to this is to employ an architecture called "King" (or any other from the published solutions) for re-executing the program with the same interleaving as far as the **Trace-Print** function is concerned. The king is a synchronization object able to influence, record and enforce interleaving.

The king architecture is used to record the order in which concurrent programs are executed and to rerun concurrent programs in the same or similar order. This architecture is used to support interactive definition of timing related tests. The tester can select a thread and advance it to a chosen program location. A King architecture contains the following elements:

- The king architecture has two modes 'record' and 'replay'. When the program is run in record mode, the king gets called when a snapshot statement is reached. The king then records the order in which snapshot operations occurred. When the program is

run in 'replay' mode, the king is called when snapshots statements are reached. The king then determines which thread to advance in order to force the recorded execution order.

- The king uses a language to record and replay an execution order. The simplest possible language used by the king is a sequence of commands each advancing a certain thread to the next snapshot instruction, e.g.,

thread 1 A; thread 1 B; thread 2 A;

- The user can use this language interactively to define the execution order.

When running multi-thread programs, each thread may write to its own trace file and when the programmer analyzes the trace file, it must clear to which thread the traced variables refer. To this end, some consistent naming scheme is required that ensures that allocates a unique name to each thread-dependent trace file and ensures that each thread writes to its own trace file.

In an example of the present invention, a method for identifying if there is no correlation between threads and traces uses bipartite matching. The method is based on the idea that initially each thread is matched to all traces. As a thread prints content, the matching is restricted to the subset of the traces that match that content. If there is no bipartite matching from the threads to the traces after a print, then the program is stopped so as to avoid executing debugger commands embedded in each of the traces at the wrong time.

Fig. 3 shows the principal steps associated with a consistent naming algorithm according to the invention for threads based on the following observations:

- A thread always has a parent thread (except for the Main Thread - i.e. the thread which starts the program).
- Since each thread executes its code sequentially, the creation time of every child thread, per parent thread, is unique.

Assumptions:

- When the system starts, there is only a single thread, called the Main Thread, or the Root Thread.
- There exists a thread bound index structure, i.e. for each thread created, there is a data structure, which holds an index counter.
- Update of the index counter is atomic.
- Thread creation time is defined as the first time a thread was created (e.g. using the new command for thread object creation in Java), rather than the time the new thread starts executing, which could be system dependent and inconsistent across program executions.

The algorithm:

As the Main Threads starts, it is named as Main, and an index structure is created for it, whose index is initialized to be 1. For each newly created child thread, at Thread creation time, do:

- Create child thread name by using its parent name as a prefix and its parent index as suffix.
- Increase parent thread's index by 1.
- Create an index structure for the child thread and initialize its index to 1.

Fig. 4 is a pictorial representation of a tree structure depicting the above-described algorithm for automatic naming of trace files as a function of a creating thread's name. Since by definition, thread

creation time is unique, and since the creation time is used to produce a unique name for every newly created thread, every thread during the lifetime of a program has a unique name. In addition, this unique name is preserved across program re-execution, as long as each thread maintains the same order of child thread creation.

Fig. 5 is a block diagram showing functionally a computer-implemented system 10 for automatically invoking a predetermined debugger command at a desired location of a single thread of a program containing at least one thread. The system 10 comprises a code modifier 11 for embedding within the program thread at the desired location thereof a utility which reads a trace file in which the predetermined debugger command has been previously embedded. A processor 12 reads the trace file during running of the program and invokes the predetermined debugger command. A file management system 13 is coupled to the processor 12 and is responsive to the embedded utility for checking whether a trace file exists, and for creating the trace file if it does not exist. A file modifier 14 coupled to the file management system 13 is responsive to the trace file being created for writing to trace file a traced value of at least one variable at the desired location of the program. A comparator 15 is coupled to the processor 12 for comparing a current value of the least one variable with a respective line in the trace file. If they are different, the comparator 15 construes the respective line in the trace file as a debugger command and invokes a debugger so as to execute the debugger command.

The program may be multi-threaded in which case there is further included a replay mechanism 16 coupled to the processor 12 for rerunning the program with identical interleaving as far as instrumentation statements are concerned. The processor 12 may further be

adapted to run the program for reading a modified trace file readable by the program wherein at least one traced value is replaced or augmented by the debugger command.

Likewise, in the event that the system is used with multi-threaded program, the file management system 13 may be adapted to create for each thread a respective trace file having a name which is uniquely defined by a name of the respective thread, thereby allowing debugger commands embedded in any of the trace files to be executed during a respective one of the threads.

The file management system 13 may also be responsive to a predetermined execution-independent naming scheme for automatically naming the trace files. To this end, the file management system 13 includes an assignment unit 17 for assigning a root name to a root thread, and a thread-bound index structure 18 for holding for each thread a corresponding index counter, which is atomically incremented upon thread creation. The assignment unit 17 is responsive to creation of a child thread for assigning a name including a prefix indicative of a name of a respective parent thread and a suffix indicative of an index counter of the respective parent thread. The assignment unit 17 is responsive to no consistent naming being possible for attempting a bipartite matching between the threads and the traces such that every thread has a trace which contains what the thread printed, and for stopping the program so as to avoid executing debugger commands embedded in each of the traces at the wrong time if bipartite matching is not possible.

A bypass mechanism 18 coupled to the file modifier 14 allows creation of the trace file to be manually or automatically bypassed so that traces are created in respect of only a subset of the threads. The processor 12 may be adapted to read the modified trace file in respect of local views of the threads only, so as to avoid a need

for synchronizing break-points in multiple threads of a multithreaded program.

It will also be understood that the system according to the invention may be a suitably programmed computer. Likewise, the invention contemplates a computer program being readable by a computer for executing the method of the invention. The invention further contemplates a machine-readable memory tangibly embodying a program of instructions executable by the machine for executing the method of the invention.

In the method claims that follow, alphabetic characters used to designate claim steps are provided for convenience only and do not imply any particular order of performing the steps.